Coordinated Atomic Actions as a Technique for Implementing Distributed Gamma Computation

A. Romanovsky and A.F. Zorzo

Department of Computer Science, Newcastle University

The intentions of this paper are to discuss Coordinated Atomic actions and to demonstrate how they can be used in a very new application area. We apply this concept to designing a particular case of the Gamma computational paradigm, i.e. distributed Gamma computation. Within our approach, each Gamma reaction is an action. We demonstrate how Gamma computation can be effectively implemented in conventional distributed message passing systems using Coordinated Atomic actions. The paper discusses our design and the benefits we gain by applying Coordinated Atomic actions: allowing as much concurrency as possible, together with guaranteeing data consistency, a better system structuring, clear separation of different system levels, and additional flexibility. This experimental design and the Java implementation allow us to conclude that Coordinated Atomic actions is a very powerful paradigm which can be used for implementing many complex systems and, in particular, software to support some parallel computational models and paradigms.

Keywords: conversations, atomic transactions, Gamma computation, coordination, Java

1. Coordinated Atomic Actions

1.1. Properties and Features

Atomic actions (or conversations) are a well-known technique intended for structuring complex concurrent systems in which several activities (processes, threads) cooperate [14, 17]. These activities (action participants) enter the action and cooperate within its scope in such a way that no information flow can cross the action border. They leave the action synchronously when all of them have agreed on the action outcome. The action execution is invisible and indivisible for the outside world. One of the most important characteristics of the atomic action is fault tolerance. All steps of providing system fault tolerance are associated with actions, so that the use of fault tolerance features is part of system design. This allows a unified, systematic policy to be imposed on providing application fault tolerance. If an error has been detected in an action, all action participants are involved in the recovery (if a participant is not able to deal with it locally). There are two kinds of error recovery: forward and backward. With backward error recovery (BER), after an error has been detected (usually by an acceptance test designed for each action), all participants are rolled back to the recovery points set when the action starts. Afterwards they can try another diversely designed alternate. Forward error recovery (FER) is associated with exception handling, so that the handlers for the same exception are called in all participants and recover the action jointly. Atomic actions can be nested, so that the execution of the system can be viewed as a tree that is dynamically updated. The main rules of nesting are simple: sibling actions cannot overlap; if a process takes part in an action, it has to take part in the father action; the action is over only if all of its nested actions are.

Atomic actions are intended for cooperative concurrent systems. The traditional way of structuring competitive systems is by using *atomic transactions* [7]. In these systems, guaranteeing the main transactional properties (atomicity, consistency, isolation and durability - ACID) is the main concern. Processes (want to) access resources (objects, files, DBs, servers) as if they were at their exclusive disposal. Although transactional support allows concurrent access, it is basically transparent for processes.

The *Coordinated Atomic (CA) action* [18, 22] concept was introduced as a unified approach to structuring complex concurrent activities and supporting error recovery between multiple interacting objects in a object-oriented system. This paradigm provides a conceptual framework for dealing with both kinds of concurrency (cooperative and competitive) [10] by extending and integrating two complementary concepts - conversations and transactions. CA actions have properties of both conversations and transactions. Conversational support is used to control cooperative concurrency and to implement coordinated and disciplined error recovery whilst transactional support maintains the consistency of shared resources in the presence of failures and concurrency among different CA actions competing for resources.

Each CA action has *roles* which are activated by action participants (some external activities, e.g. threads, processes) and which cooperate within the CA action scope. Logically, the action starts when all roles have been activated (though it is an implementation decision to use either synchronous or asynchronous entry protocol) and finishes when all of them reach the action end. The action can be completed either when no error has been detected or after a successful recovery or when the recovery fails and a failure exception is propagated to the containing action.

External objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among these actions and that any sequence of operations on these objects bracketed by the CA action start and completion has the ACID properties with respect to other sequences. CA action execution looks like atomic transactions for the outside world. One of the ways to implement this is to use a separate transactional support that provides these properties. A number of such schemes are discussed in [7]. They offer the traditional transactional interface, i.e. operations `start`, `abort` and `commit`, which are called (either by CA action support or by CA action participants) at the appropriate points during CA action execution.

The state of the CA action is represented by a set of *local objects*; the CA action (either the action support or the application code) deals with these objects to guarantee their state restoration (which is vital primarily for BER). Local objects are the main means for participants to interact and to coordinate their executions (although external objects can be used as well). The two existing kinds of local objects, shared and private, are treated differently. The former are intended for role cooperation, and their consistency is provided on the application level rather than by CA action support (one of the ways is to design them with monitor semantics [9], e.g. as Ada 95 protected objects). Private local objects are used by individual action participants and represent their internal states.

CA actions can use both BER and FER as well as their combination. In this respect they are similar to conversations. When FER is used, the action body is the exception context in which exceptions can be declared, exception handlers are associated with each role and

exception resolution is used to resolve several exceptions raised by several roles; the failure exception is used to inform the containing action when the action fails to recover (in this case the atomicity property requires that all changes made during the action execution be reversed and operation `abort` done on external objects). A general object-oriented framework for introducing FER into CA actions is discussed in [20]. This paper clearly shows why resolution should be used and why it is vital for many distributed systems.

The CA action concept allows degradation by using several levels of the action outcome (*multiple outcomes*). This can be used if an action is not able to guarantee the required service but can provide a service of a lower quality: one of the outcomes is returned by the action depending on the quality of the results it has been able to produce. The easiest way of informing the containing action about these degrading outcomes is by using additional interface exceptions.

In the last two years we have gained considerable experience in designing CA action schemes using Ada 95 and Java [18-21]. Our implementations are presented as sets of re-usable components (classes, generic procedures, packages, objects) and of templates and conventions which should be followed by programmers. These schemes differ in the ways of packaging actions and participants (e.g. actions can be associated with packages or classes), use different kinds of recovery or different concurrency control policies for external objects; some of them are single-computer, others are distributed, with different kinds of action component distribution, etc.

1.2. Structuring Complex Systems

Most structuring techniques are intended for organising the static structure of the system: modules, objects, classes, etc. [16]. However we agree with the researchers pointing out that component interaction should be structured as well as the static system organisation [11, 15]. There are many reasons why we believe that this is vitally important. One of them is that the dynamic system structure can be much more complex than the static one. Using atomic actions for structuring the dynamic aspects of the system essentially simplifies the system design by hiding complexity inside actions, by system layering (action nesting), by providing a unified way of tolerating faults (fault tolerance features are attached to actions). Moreover, this facilitates reflecting upon the system, proving its correctness (one of the reasons is that using atomic actions can essentially reduce the number of interleavings in the concurrent program), introducing parallelism into system execution, ensuring correct access to shared data. Many researchers [3, 11, 15] believe that using atomic actions improves and facilitates many stages of the software lifecycle: design, validation, maintenance, etc.

The same considerations are applicable to CA actions: they facilitate the design of complex systems by hiding the complexity of the system concurrent/dynamic behaviour inside CA actions and by imposing a multi-level design on the description of the system dynamic behaviour. Moreover, CA actions have some unique properties not found in atomic actions: first of all, those of atomic transactions (the restricted applicability of atomic actions with respect to accessing or competing for shared data is discussed in [8]). CA actions are intended for designing and structuring complex systems in which components can both cooperate and compete (this cannot be provided separately either by conventional atomic transactions or by conversations).

Providing a systematic and structured framework for achieving fault tolerance is a very important characteristic of CA actions. In this paper we are going to apply CA actions to designing and structuring a complex system, and although the original application does not have any special dependability requirements, we extend this system by formulating very reasonable fault assumptions, so that our design tolerates these faults.

2. Distributed Gamma Computation

The *Gamma model* [1, 2] was proposed to allow treating computation as a global evolution of a collection of atomic values interacting freely. This approach can be introduced intuitively through a chemical reaction metaphor. The unique data structure in Gamma is the multiset which can be seen as a chemical solution. A simple program is a pair `(reaction condition, action)`. Execution proceeds by replacing the multiset elements satisfying the condition by the product of the action. The result is obtained when a stable state is reached, i.e. when no more elements satisfy the reaction condition. An example is a program calculating the maximum element of a non-empty set:

```
max: x,y -> y  <=  x<y
```

where two elements react and are replaced in the set by the maximum of the two. The essential feature of the Gamma is that individual values may interact and produce new values in a completely independent way. This paradigm does not introduce unnecessary sequentiality as opposed to the traditional paradigms which are overconstrained as compared with the logic of the problem to be solved. We refer the interested reader to [1, 2] (where new references can be found) for a more detailed discussion, formal description and the experience gained in numerous applications developed using this paradigm.

Distributed Gamma computation (the *DSGamma model*) [4, 5] is based on the Gamma paradigm. It uses the number (integer) addition operation. The main idea is to offer a particular distributed model (in a sense, a particular case of the original Gamma model) that distributes the (*global*) multiset in such a way that it is stored and dealt with distributedly as a set of *local* multisets. This requires taking more care of their consistency. The purpose is not to offer a complete model extension but to discuss a small but representative example for reverse engineering mainly [5]. Several changes have been made in the DSGamma model: the multiset can be updated either by chemical reactions or when a user inserts new values; the multiset is distributed, which means that it is presented as a set of local multisets located on different distributed nodes. The assumption is that there is a set of host computers connected via a network that can exchange messages and keep one local multiset each. A new user can join the system through any host computer, in which case a local multiset is created. The model does not define on which computer the chemical reactions are executed or whether there are other computers in the systems apart from the host (users') computers. As soon as at least two integers are present in the global multiset, the reaction occurs. The reaction consists in randomly removing two integers and in inserting their sum in the global multiset (into a randomly chosen local multiset). DSGamma is a way of dealing with the Gamma model using distributed systems. In a sense, DSGamma is a way of designing and implementing the original (slightly modified) Gamma model on the conventional hardware and software. We believe this approach is promising because it can be easily applied to

designing new Gamma applications on the web when applets are loaded into host computers.

We have found this example very interesting for applying CA actions to designing concurrent, parallel and distributed systems. It is representative enough because it has both cooperation and competition; there are complex distributed shared data that need to be consistent; and finally, this application requires as much parallelism as possible. We have decided to design the DSGamma system using CA actions and by doing this to show how Gamma computation can be effectively implemented on conventional distributed message passing systems.

3. System Design

3.1. System Analysis, Components and their Functionalities

We start designing a system implementing DSGamma computation by reformulating its most important characteristics in the following way. Reactions can be executed in parallel provided there are enough data for them in the multiset and that the consistency of these data is guaranteed. It is assumed that all reactions are atomic (we will show how to guarantee this, which is not a simple task, in distributed systems when we deal with implementing this computational model using conventional software). Chemical reactions are costly, and thus their execution should be distributed. But then we have to assume that message passing between computers is much cheaper. Another reason for distributing the computation (i.e. the chemical reaction) is that as much parallelisation as possible should be allowed. One more assumption is that the multiset is huge, so it makes sense to distribute it and keep it as a set of local multisets.

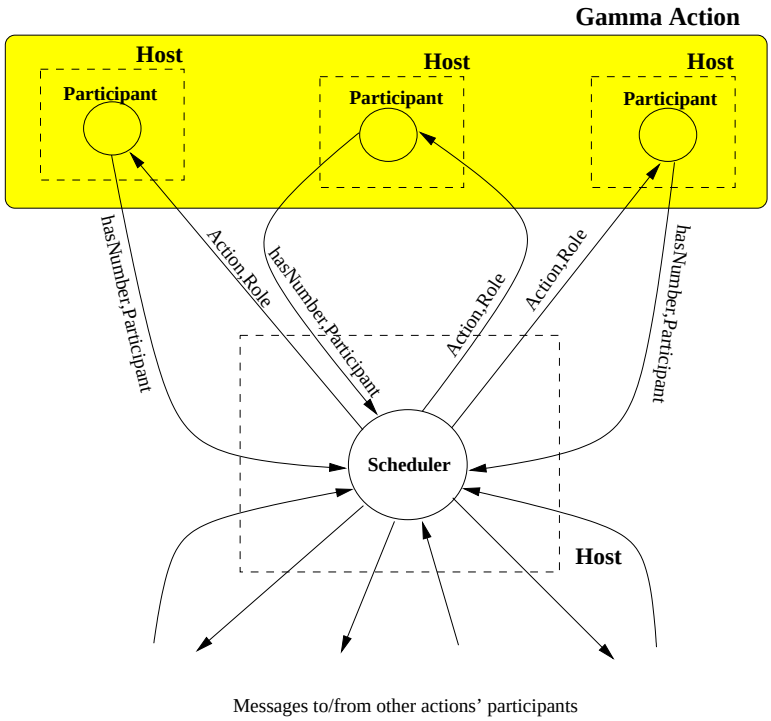

Figure 1. Structure of DSGamma system

Our system is composed of a set of *participants* (located on different hosts), a *CA action scheduler* (located on a separate computer) and a set of CA actions in which participants are involved (Figure 1). A participant starts when the operator presses a button on a host computer for the first time. The global multiset is scattered through all participants, so that each participant has a local multiset (queue) keeping some part of the global multiset. CA actions are activated dynamically to execute Gamma computation. Each action has three roles: two producers (each of them has a number in its local multiset) and a consumer that sums them up (the chemical reaction). CA actions enclose the interactions between participants on the level of Gamma computation. Local multisets, as well as the global multiset as a whole, are viewed by CA actions as external objects. The CA action scheduler receives information from all participants about any new number they have in their queues and starts a new action with three roles when there are two new numbers in local multisets. There can be as many actions active concurrently as there are pairs in all local multisets at a given time. For example, it is allowed to have several active actions in which the same participant takes part as a producer (if there are several numbers in its local multiset) or/and as a consumer. This allows a better parallelisation of Gamma computation.

Our system has two design levels. The first level represents the information exchange between computers (the participants and the CA action scheduler). This is the level on which the execution of CA actions is organised and scheduled (sometimes we say that this level glues actions together); it could be designed using CA actions but we have not done so because we think that they are not quite suitable for designing this sort of underlying scheduling support. The second level of our design is the level of Gamma computation itself, where the interactions between participants and the access to external objects are executed. On this level the numbers are passed between different local multisets and summed up. We have used one of our CA action schemes [21], the Java centralised one, to implement the system. Depending on the hardware peculiarities or on some a priori knowledge about the application (e.g. the rate of users joining/leaving the system), other algorithms, less centralised ones among them, can be used for designing the first level. One can think of connecting all hosts in a virtual ring or using broadcast to match several participants ready for the chemical reaction. It was not our intention to investigate this direction any deeper because we decided against assuming any additional knowledge about the system and offering the best partial solution. The purpose of our research was to design DSGamma computation using CA actions. The design of the second level is general enough; it can be used with any kind of scheduling support that has been chosen to implement the first level of the system.

3.2. Action Participants

In our design each participant has two service threads on the first level. The first thread `TGetNumbers` receives numbers typed by the operator, and inserts these numbers into the local multiset `ParticipantQueue`. After the number has been inserted in `ParticipantQueue`, the thread sends message `hasNumber` to the CA action scheduler informing that a new number has been typed and stored in `ParticipantQueue` (this means that the participant is ready to execute a chemical reaction, which is a CA action in our design). The second thread `TExecuteActions` receives messages from the CA action

scheduler that contains a pointer to the action that the participant must join, and the role that the participant must execute in that action.

When thread `TExecuteActions` receives such message, it starts a new thread to execute the role in the action. This new thread is called `TConsumer` if the participant has to execute the Consumer role in the action, or `TProducer` if it has to execute either role `FirstProducer` or role `SecondProducer` in the action. After `TConsumer` has finished the execution of its role inside the action, it sends message `hasNumber` to the CA action scheduler signalling that another number has been inserted in `ParticipantQueue` (Figure 2). Threads `TConsumer` and `TProducer` are destroyed as soon as they have finished their role execution in an action (Figure 3).

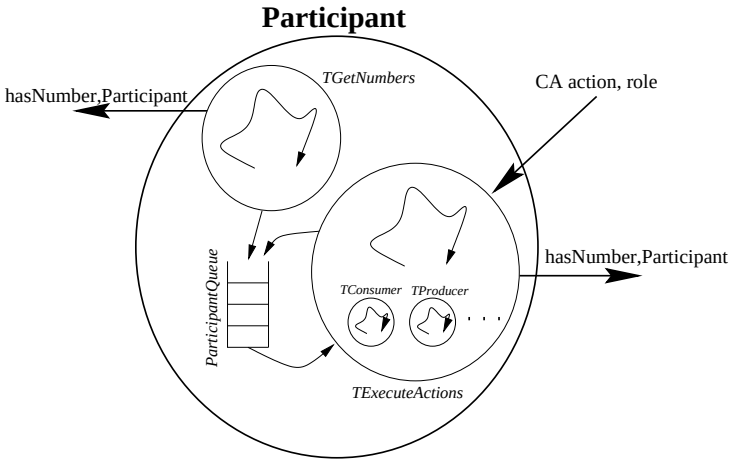

Figure 2. System participant

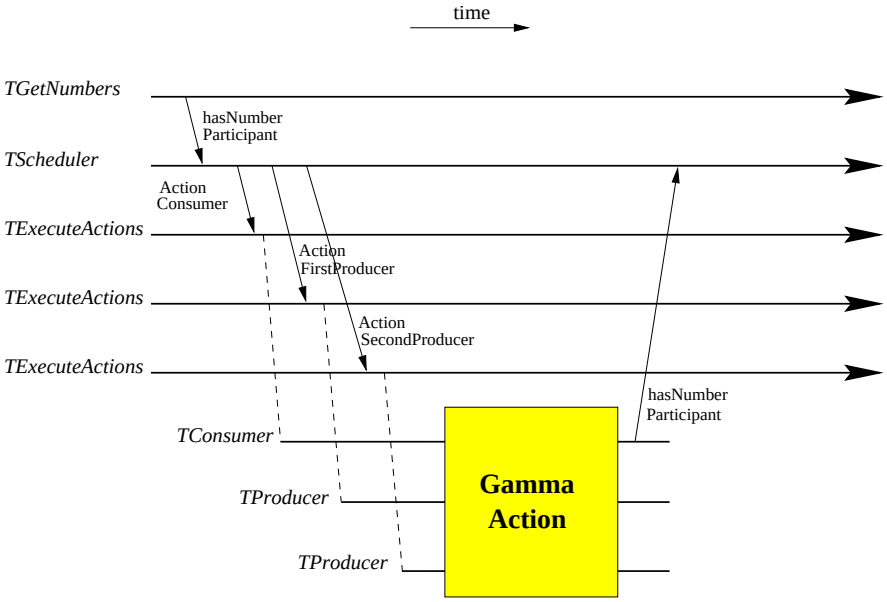

Figure 3. Creation/destruction of threads

3.3. CA Action Scheduler

The CA action scheduler (one for the entire system) triggers the creation of all actions and matches three participants to execute an action. Two of them that have numbers (and are

ready to take part in an action) and the third one which be the consumer of these two numbers (it sums them up and puts the result into its `ParticipantQueue`). The CA action scheduler has a service list of all participants, `ParticipantsList`, that contains two items of information for each of them: the address of the participant and how many numbers are stored in its `ParticipantQueue`. When the CA action scheduler receives message `New`, it inserts a new participant in the list. When the CA action scheduler receives message `hasNumber`, it increases the number of integers that this participant has in its `ParticipantQueue` (Figure 4).

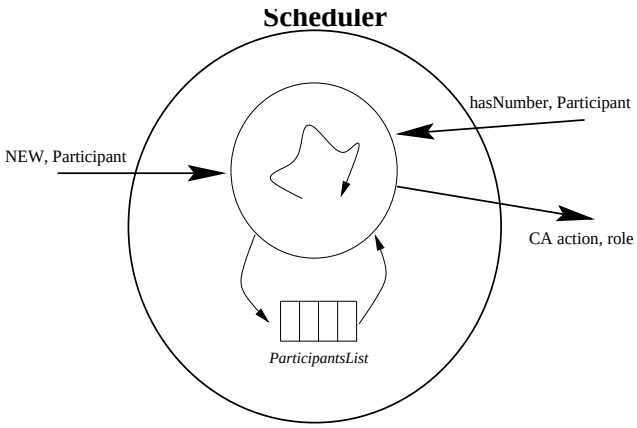

Figure 4. CA action scheduler

When a participant has been chosen to be a producer in a Gamma action, the CA scheduler decreases the number of integers of this participant by 1, so that it does not have to inform the CA scheduler that it passes an integer to the consumer and that the producer's multiset has one number fewer when the action is over. The scheduler decreases this number in an optimistic way when it chooses the producer and sends it the reference to the action which the producer is to enter.

The execution of the CA action scheduler consists of receiving those messages and of randomly choosing the consumer when it has two producers ready to take part in an action. It sends a message to the participant and tells it that it should take part in a particular action (the name is passed) as a consumer. The CA action scheduler uses the same list to choose two participants that have numbers to be summed up (that means that they have numbers in their multisets `ParticipantQueue`). The termination of the Gamma reaction is detected by the scheduler; the reaction is over when there are no more users in the system and there is only one number in the multiset.

3.4. `GammaAction`

The CA Action for the Gamma reaction itself is straightforward. It has three roles: `FirstProducer`, `SecondProducer`, and `Consumer`. The producers take the numbers from their multisets `ParticipantQueue` and send them to role `Consumer` which sums them up and inserts the result into its multiset (Figure 5). The Gamma action, as well as the other actions in our design, use all the main features of the CA action concept. CA action `GammaAction` only starts when all participants are ready to execute their roles in the action, i.e. they synchronise upon entry. The termination of the action occurs only when all participants have finished the execution of their roles and no exception has been raised, or

when an exception has been raised, and the participants have recovered the action and agreed on the outcome. Participants use local objects as means of inter-role communication, e.g. roles `Producer` access their multisets and use a local object to send their numbers to the `Consumer`. Each participant has access only to its multiset and to the local objects of the action.

The pre-condition of this action is as follows: there are two numbers in the global multiset. The post-condition is that these two numbers are not in the global multiset but there is a new number equal to their sum.

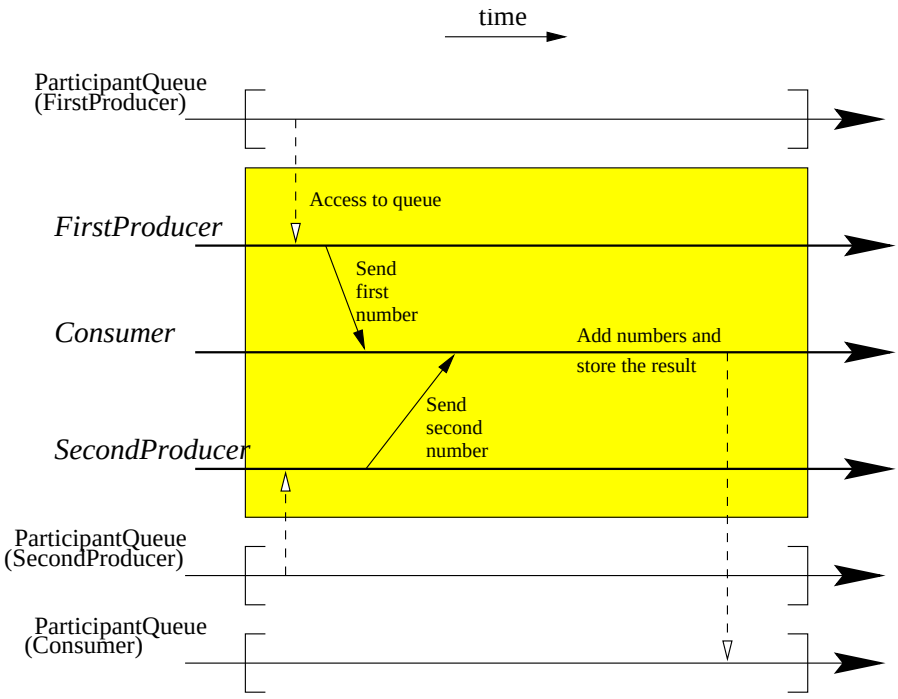

Figure 5. Gamma CA action

Local multisets `ParticipantQueue` are external objects in our design and can be accessed only within CA actions. Their consistency and integrity are guaranteed by CA action support in such a way that several actions can take numbers from the same object and add new numbers in it concurrently. First of all, as we have mentioned, they have monitor semantics so only one number can be taken or inserted at a given time. Our particular implementation uses a simplified concurrency control: it allows us to lock just one number in the producer local multiset but not the entire queue. This approach guarantees maximum concurrency, because we do not restrict the access to that multiset to one action at a time, so in a sense we view each element of the multiset as being an object itself.

Several actions can be active in the system at the same time, and each participant can be involved in many actions playing roles of producers or/and consumers. The CA scheduler triggers the creation of an instance of `GammaAction`; it does not wait for the end of this action and is ready to trigger the creation of a new action if there are two new integers in the system.

The CA action scheduler involves participants in CA actions. This design is very general, so we assume that there can be a set of hosts on which the instances of CA actions can be instantiated, and that it is up to the scheduler to set their location.

3.5. `InsertNumberAction` and `FinishAction`

Apart from `GammaAction`, we employ two additional service CA actions: action `InsertNumberAction` for inserting a new number typed by the user into the local multiset and another action `FinishAction` for transferring the entire local multiset to another host when a user decides to leave the system. These actions are executed by thread `TGetNumbers`. When a new number is entered by a user, thread `TGetNumbers` executes a role inside action `InsertNumberAction`. When the user wants to finish his/her participation in the Gamma system, this thread enters `FinishAction`.

Action `InsertNumberAction` has just one role which is responsible for inserting the number in the local multiset `ParticipantQueue`. This CA action has the properties of a simple transaction. The user produces a number which is passed to this action to be inserted into the participant multiset. One action inserts only one number. On any host this action can be executed concurrently with action(s) `GammaAction` but not with another `InsertNumberAction`, because the thread responsible for activating this action is also responsible for servicing the user, i.e. the thread executes the following steps in a loop: i) gets number `X` from the user, ii) activates action `InsertNumberAction` and sends number `X` as the action parameter. The post-condition for this action is that a new number is inserted in the local multiset.

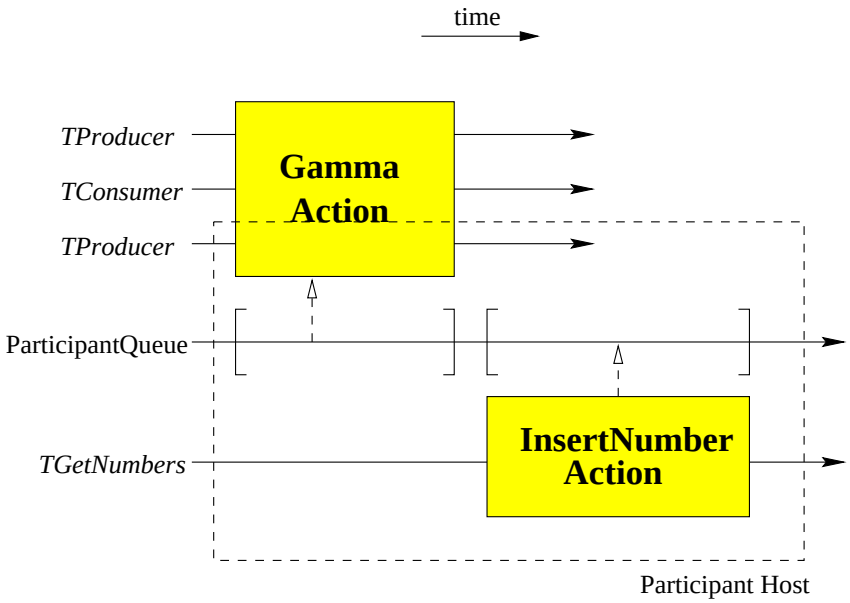

Figure 6. Competition between actions for external object `ParticipantQueue`

Action `FinishAction` has two roles: the first one is executed by thread `TGetNumbers` and the second one is executed remotely by another participant. This second participant is chosen randomly by the CA scheduler from among all participants that are present (except for the participant that wants to leave the system). `FinishAction` transfers all integers from the local multiset of the participant that wants to finish its execution to the local multiset of another participant. When a participant decides to finish, our support informs the scheduler of that, and the scheduler chooses another participant to execute a `FinishAction` together with the first one. When `FinishAction` is completed, the participant which has received new numbers informs the scheduler of new integers in its multiset. The participant waits until all active actions `GammaAction` in which it is involved

are completed, and only afterwards enters action `FinishAction`. Once the scheduler has chosen the participants to execute `FinishAction`, it continues its execution, and the participant willing to finish will not be chosen by the scheduler again to execute any other action. The post-conditions of this action are straightforward: all numbers from the local multiset are transferred to another randomly chosen local multiset, the numbers are unchanged and the global multiset remains unchanged.

Figure 6 shows the competition between two actions for external object `ParticipantQueue`: although their accesses to the object overlap, on the logical level these accesses are serialised, and the object consistency is guaranteed.

3.6. Tolerating Faults in `GammaAction`

The original DSGamma paradigm assumes that there are no faults in the system. We decided to include some fault tolerance in the system to demonstrate how CA actions can be used and to make the system more realistic. We assume that faults can happen in the chemical reactions, and that a predefined exception `ReactionException` is raised in the thread executing the reaction. We assume that nodes and channels are reliable (which means that their fault tolerance, if required, is implemented transparently for our system by the underlying support).

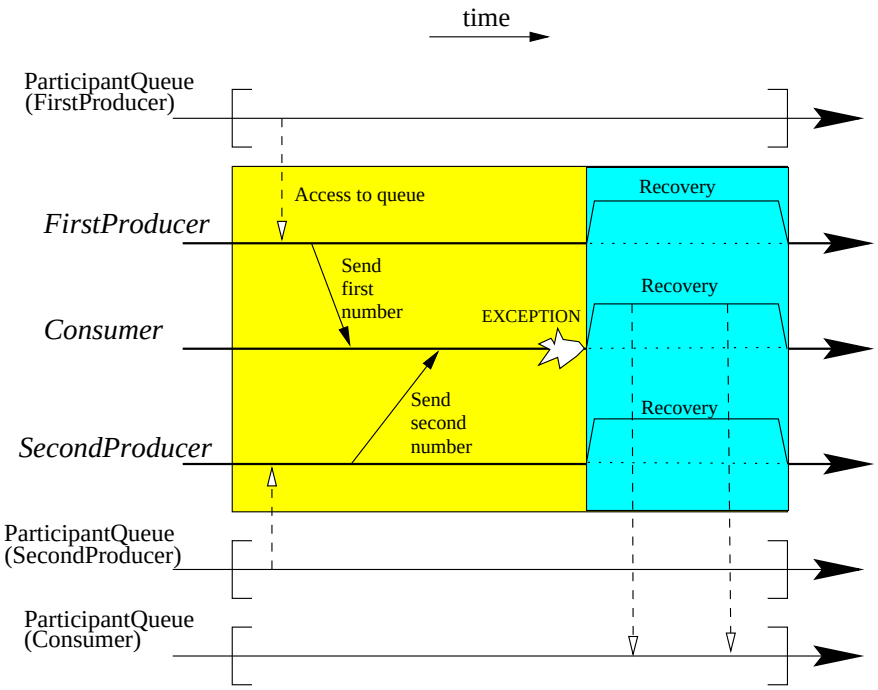

Figure 7. Tolerating faults in chemical reactions

In accordance with the CA action concept we attach fault tolerance to each action. After exception `ReactionException` has been raised, CA action support interrupts all action participants (this is a pre-emptive CA action scheme - [18]) and calls the handler for this exception in each of them because all roles are to be involved in cooperative action recovery (Figure 7). Our design decision is to recover the action in the following way: the consumer, which keeps both integers when the reaction fails, inserts them into its local multiset, and the producers complete the action as if nothing has happened. As we mentioned in Sections 3.2 and 3.3, the scheduler changes the numbers of integers in producers' local multisets in an

optimistic way, when it chooses the producer and sends the action to it to participate. If a fault happens during the action execution, then the consumer recovers the system within action `GammaAction` as explained before, but in this case two new integers appear in the consumer's local multiset. We use a special outcome of action `GammaAction` to inform thread `TConsumer`, whose responsibility is to inform the scheduler about these new numbers (by calling method `hasNumber` twice).

Actions `GammaAction` are atomic with respect to the faults of the chemical reaction: exception handlers guarantee "nothing" semantics for the global multiset (although local multisets are modified during this recovery). Using multiple outcomes allows us to implement a very cheap local recovery which is executed on the consumer host only.

4. Implementation in Java

We have implemented DSGamma system using Java programming language [12] and Remote Method Invocation (RMI) API [13]. We use one of our ready-made Java CA action schemes which was designed for our first set of experiments with a Production Cell case study [23]. This is a centralised CA action scheme: each action has an action manager. In this scheme an action object is created for each action, so that action roles are its private methods. Each action can have local objects (inside the action object) that are used by all roles for communication and synchronisation.

We have implemented the CA action scheduler as a remote object that can be accessed by participants to inform it when they are joining the system, when they are willing to leave the system, and every time they have a new number inserted in their local multisets. The CA action scheduler object has the following interface:

```
public interface CAScheduler extends java.rmi.Remote
{
  public void newParticipant(Participant newPart)       throws RemoteException;
  public void hasNumber (Participant part)              throws RemoteException;
  public void endParticipant(Participant part)          throws RemoteException;
}
```

Method `newParticipant` includes a new participant in the CA scheduler list; in particular, the caller has to send a reference to its remote object that can be accessed by the CA scheduler. Method `hasNumber` is used by participants to inform the CA scheduler each time a new number is inserted in its local queue. When a participant has numbers in its queue, the scheduler can select it to perform a producer role in some action `GammaAction`. Method `endParticipant` is used by participants to inform the scheduler when the participant is willing to finish its execution: the scheduler chooses another participant to which the numbers from this participant are to be transferred.

The participants are implemented as remote applets that can be accessed by the CA action scheduler or by other participants. Each participant has a queue, or a local multiset object (which is created by this participant and located in the same place where it is) that stores the numbers of its local multiset. This object implements its operations (`put`, `get`) using a monitor style approach (all methods are Java synchronised methods [12]). Each participant also has a list of `GammaAction` objects in which it always performs role `Consumer`, i.e. when a CA action in that participant is activated, then the participant containing the action

takes part in the action as the consumer (the CA action scheduler will set that). The participant applet has the following interface:

```
public interface Participant extends java.rmi.Remote
{
  boolean sendGammaAction(Participant where, String role, Integer caId)
                                  throws java.rmi.RemoteException;
  boolean remoteGammaAction(String role, Participant part, Integer caId)
                                  throws java.rmi.RemoteException;
  boolean sendFinishAction(Participant where, String role)
                                  throws java.rmi.RemoteException;
  boolean remoteFinishAction(String role, Participant part)
                                  throws java.rmi.RemoteException;
  void remoteQueuePut(int num)    throws java.rmi.RemoteException;
  int  remoteQueueGet()           throws java.rmi.RemoteException;
  boolean remoteQueueIsEmpty()    throws java.rmi.RemoteException;
}
```

where method `sendGammaAction` is used by the CA action scheduler to inform the participant that it has to execute role `role` in `where` (parameter `caId` is a unique identifier of the action the participant has to enter, which guarantees that only the participants with the correct identifiers execute the action);

method `remoteGammaAction` is used by other participants that want to execute role `role` in the action located in this participant (the willing-to-execute participant sends a reference to itself; then the action can access its local queue);

method `sendFinishAction` is used by the scheduler to inform the participant that it has to execute action `FinishAction` together with another participant;

method `remoteFinishAction` is called by another participant to execute action `FinishAction` in this participant;

methods `remoteQueuePut`, `remoteQueueGet`, `remoteQueueIsEmpty` are used by a participant when executing action `GammaAction` remotely. It provides access to the local queue of this participant.

Figure 8 shows object `GammaAction` and the roles of this action. The CA action object is composed by a set of internal objects, used only by the roles of the action in order to exchange values, e.g. communicate; a set of external objects that the roles access atomically; a manager that is responsible for recovering the action from possible failures and for the participants' synchronisation on action entry and exit; and the roles that the participants execute. In order to execute a role in an action, participants must be informed which action and role they have to execute. Such information is provided by the CA action scheduler using participant method `sendGammaAction`. Once participants have received the information on which action and role they have to execute, they activate the action by calling method `inAction` in the action object which sends information about the participant local queue. These local queues are external objects for the CA action manager, and binding them to the CA action is done dynamically. Method `inAction` executes the functionalities of the CA action manager discussed above.

Our system allows tolerating faults in the chemical reactions by using FER (Section 3.6). The Java implementation follows the general approach proposed: when an exception is raised in one of the roles, this exception is caught and the role informs the CA action manager. The manager interrupts all other roles using the Java ability to interrupt threads asynchronously (generally speaking, several roles can raise exceptions at nearly the same time which is not the case in our system). Note that we do not need exception resolution.

This is why the manager raises exception `ReactionException` in all participants, so that they start cooperative recovery. As we have explained, the roles return a special outcome to inform the participating threads of the fact that the action has produced a degrading service, in which case thread `TConsumer` informs the scheduler not about one new number (as happens when there is no fault inside the action - Section 3.2) but about two new numbers inserted into the consumer queue.

As we have mentioned we use an existing CA action scheme in our implementation. We re-use it without serious modifications; the main adjustment we have made is providing a fine-grained concurrency control for multisets. This simple control is implemented using a counter associated with each local multiset and kept on the scheduler side. So, the next action that consumes the number from a local multiset can be scheduled only if the counter is positive. As a result, when the action is executed, there is no need for it to lock/unlock the multisets involved.

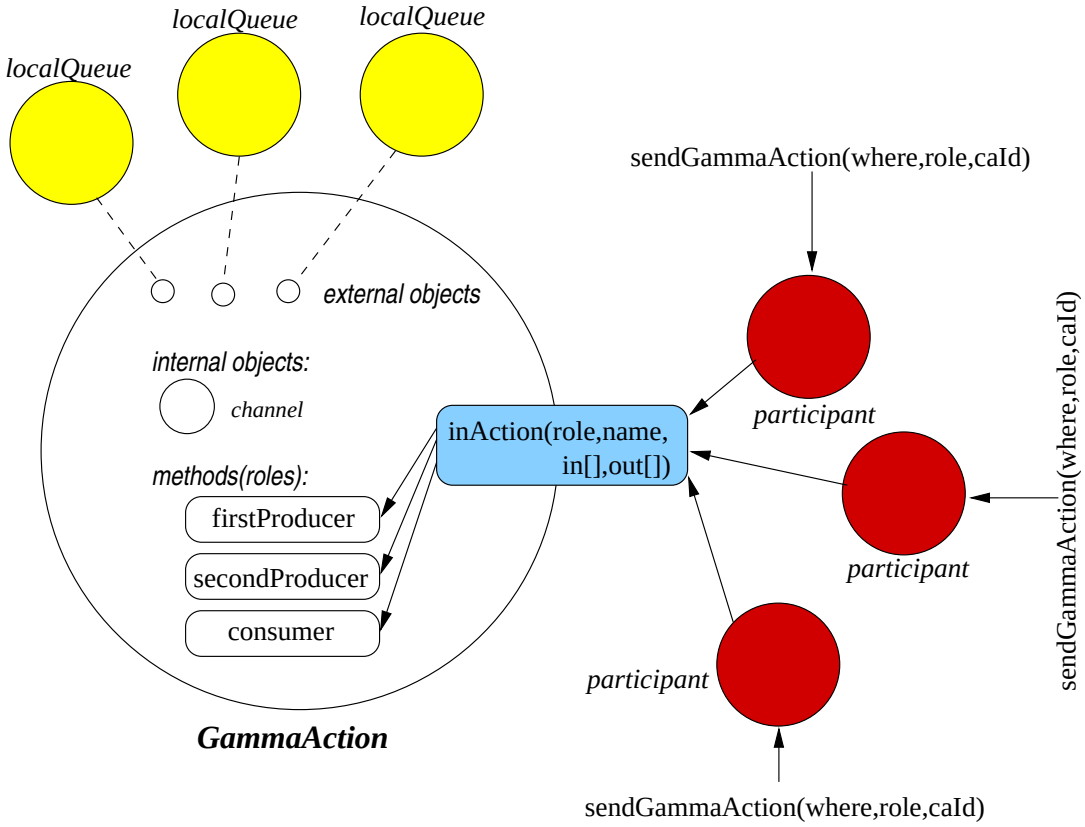

Figure 8. Object `GammaAction`

We have decided to locate CA actions in participant hosts rather than in the scheduler. This allowed us to avoid overloading the CA scheduler node with the execution of many actions and made the execution of the second level of our system completely decentralised and distributed.

5. Discussion

We believe that it is very important that we completely separate the design of DSGamma computation (consisting of the chemical reactions only) from the 'gluing mechanism' that

initiates Gamma computation when and where necessary. This mechanism is supporting software which can be designed using CA actions as well.

Our approach allows us to implement DSGamma computation in a very efficient way because we treat multisets as external objects for CA actions and implement a very simple, conflictless and optimistic technique for dealing with them: it is guaranteed that actions never wait for access to multisets.

There are many reasons why we are confident that the added price we have to pay for using CA actions is not high. We use a specially-adjusted CA action scheme, decrease the number of messages passed between nodes (for example, the producer sends one message to the action object both to enter the action and to pass a remote reference to its local multiset) and use application-specific concurrency control that allows any number of actions `GammaAction` to be concurrent. Another reason why this system is effective is that we use a very optimistic approach to provide fault tolerance. This is why we do not spend any time on providing fault tolerance if there are no faults in chemical reactions (e.g. we do not use checkpointing). Moreover, the recovery after the reaction fault is local. CA actions allow us to use a very clean way to recover the system employing the concept of multiple outcomes.

Our system works effectively if the cost of the chemical reaction is much higher than the cost of providing CA actions: action object creation, participants' synchronisation upon entry and exit, access to external objects involved in the action. To improve the performance of our system, we have made several choices that lower the price of CA actions. We are also aware of several other ways of adjusting CA actions for this particular application and of improving the gluing level.

Our design is applicable if the volume of the global multiset is much larger than the volume of special data kept by the scheduler; this can be the case if there are many numbers (elements) in local multisets and if the volume of each reacting element is large (i.e. when the system is not used for simple addition of numbers but for much more complex operations on large data structures).

Our approach is very flexible. Thus, for instance, at a later stage of our design, when we realised that we needed an always-alive participant to keep the resulting value after all users have left the system, it was easy to create one on the scheduler node. One simple modification could be using a different policy for transferring the local multiset of the leaving user: one local multiset can be chosen randomly for each number but not for the entire multiset. To do this, we have to create as many new actions `FinishAction` as there are numbers in the leaving host. Another likely and straightforward extension could be to re-design the system and to execute the chemical reaction not on the consumer's node but on the producer's node, or even on a fourth, unrelated node (that could be chosen either randomly or using another policy, e.g. a pool of 'calculating computers' can be used).

Our analysis shows that our design solutions can be applied to the original Gamma paradigm [1, 2] and that the design proposed is general enough to allow effective and clean implementation. To do this we need another service, to deal with several global multisets. Our scheduler can provide this naming service if a list of multisets with their locations is included. Afterwards we can distribute global multisets and parallelise Gamma reactions by

forking as many CA actions as required. If the above mentioned assumptions on the volume of data in multisets and the length of chemical reactions hold, our approach can be effectively used in spite of the fact that it is centralised.

6. Lessons Learnt

The starting point of our design was that we wanted to guarantee that the basic operations of DSGamma computation (i.e. chemical reactions) are atomic, indivisible, invisible and reliable. These are the main requirements for many computational models and this is where we believe the CA actions can be successfully applied. These operations should be executed as much as possible in parallel, guaranteeing the data (value) consistency, which is again the essence of the DSGamma paradigm. This is why providing the ACID properties is the most general solution. These are difficult to provide in complex distributed systems, and the designer of any particular implementation may not even realise their generality by doing it in an ad hoc way.

CA action support itself takes care of providing many important properties: consistency of the data shared among competing reactions, action starting/finishing, prevention of information smuggling, tolerating faults, etc. It is important that this support is re-usable and that our research offers a wide range of CA action schemes.

Our approach allows us to design a two-level system with a clear separation of levels functionality, with complex concurrent behaviour hidden inside actions and levels. This can facilitate, for example, reflecting upon the systems and analysing and proving their properties (see, for example, [6]). In case of the DSGamma the level of CA actions is nearly trivial because the actions, their functionalities and their pre- and post-conditions are straightforward.

Although the DSGamma model does not have high dependability requirements, our previous experiments [23], together with implementing fault tolerance for this model, allow us to conclude that using CA actions imposes a very disciplined and unified way of providing fault tolerance through the entire system.

We believe that applying CA actions allows us to achieve a maximum degree of parallelism: this parallelism is restricted by only one inevitable reason, i.e. guaranteeing the global multiset and local multiset consistency. We can conclude that the CA action approach is the most general model suitable for designing many complex concurrent and distributed systems in which there is a restriction on the parallel execution of system parts/components caused by using the same data. Generally speaking, there can be causal restrictions, when one part waits for another to be finished and to produce the required data, and consistency restrictions, when several parts would like to have as much parallelism as possible when accessing the same data. CA actions are able to deal with both of these problems.

We believe that CA actions should be used as a special technique for structuring complex concurrent systems of different nature. And although for some systems the overall performance can decrease, we believe that the benefits gained by getting a simple design, using re-usable components, allowing as much parallelism as possible, using a disciplined way of providing system fault tolerance, etc. can be more important either in the long run or

for fast prototyping. For example, the latter can allow users to start experimental programming earlier, using new and exotic computational models.

Our analysis shows that the conventional design does not usually support the structuring of concurrent system dynamic behaviour and that conventional systems do not have any features for describing both component cooperation and competition in complex systems. Although the functionalities provided by systems with CA actions and without them can be nearly the same, the designs differ a lot. Conventional techniques do not allow imposing structure on the component cooperation which is usually flat and consists only of component synchronisation (e.g. message passing). In our approach, cooperation has two levels: the CA action level (potentially CA actions allow any numbers of levels of structuring) and the action scheduling level. Moreover, within our approach providing data consistency is not system or component designers' responsibility.

There are obviously additional (mainly performance) overheads for using CA actions: additional messages and increasing overall system synchronisation. One can design a faster system by taking advantage of some low level knowledge of the application and by breaking information and concurrency encapsulation imposed by CA actions. It is the same with any structuring techniques and with any sort of disciplined programming.

7. Conclusions

This paper describes one of our experiments with using CA actions for designing complex and very unusual applications. Our previous experiments dealt with designing a complex distributed control system (the production cell) and are believed to have been quite successful. As they were carried out, we realised how CA actions could be used for designing pipe-line applications in such a way that the maximum concurrency (parallelism) is provided together with guaranteeing the consistency of the system state. In this respect our conclusions from the DSGamma experiment are very similar although we applied CA actions for a completely different kind of systems.

We believe that CA actions can be used for implementing computational models of many parallel paradigms and architectures (e.g. data-flow, pipe-line, event-driven, functional programming, etc.) when traditional hardware and software is used. In paper [1] a number of languages and formalisms bearing similarities with the Gamma computation are surveyed. Our analysis shows that all of them (i.e. *associons*, Unity, Linda, Linear Objects - see [1] for their brief description and references to the original papers) can be implemented using CA actions. The same is true for several extensions of the basic version of Gamma (such as schedules and local linear logic - [1]). The requirements common for all of them are: achieving maximum parallelism of basic operations, guaranteeing consistency of data accessed by these operations and atomicity of these operations. These are exactly the properties that CA actions provide. We believe that the concept of CA actions is very general and can be employed in many different applications in which parts of systems can either compete for resources or cooperate to achieve some joint purposes. This paper demonstrates how this concept was used to implement a new computation paradigm and discusses the lessons we learnt while doing this. We believe that this implementation is quite successful:

the resulting system has a clean structure and all the required properties, it is quite flexible and its essential parts (CA action support) are re-used.

Our DSGamma system has been implemented over the Internet and the participants are applets that can be downloaded from the scheduler side. This experience allows us to conclude that CA actions can be used for designing complex web applications in which some parts of the cooperation should have the atomicity property with respect to the rest of the system, with a guarantee of consistency for the data shared by these parts (designed as CA actions) and the rest of the system (which may consist of a number of CA actions in its turn).

Acknowledgements. We would like to thank G.DiMarzo and N.Guelfi (EFPL) for introducing us to the idea of Gamma computation and helping us with understanding the distributed Gamma addition model. We are grateful to our colleagues at the University of Newcastle upon Tyne: B.Randell, J.Xu, R.Stroud and I.Welch for the inspiration we have derived from investigating the concept of CA actions together with them. This research has been supported by the ESPRIT LTR Project 20072 on Design for Validation. A.F.Zorzo is also supported by CNPq (Brazil) under grant 200531/056.

References.

[1] J.-P. Banâtre and D.L. Métayer, Gamma and the chemical reaction model: ten years after, in: J.-M. Andréoli, C. Hankin, and D.L. Métayer, eds., *Coordination programming: mechanisms, models and semantics* (World Scientific Publishing, IC Press, 1996) 3-41.

[2] J.-P. Banâtre and D.L. Métayer, Programming by Multiset Transformation, *Com. of ACM* 36, 1 (1993) 98-111.

[3] E. Best, *Semantics of Sequential and Parallel Programs* (Prentice Hall, 1996).

[4] G. DiMarzo, Distributed Gamma-like Addition of Integers, Internal Tech. Report and Web Page (http://lglsun.epfl.ch/Team/GDM/java/DSGamma/DSGamma.html), Softw. Eng. Lab., EFPL, Switzerland, 1997.

[5] G. DiMarzo and N. Guelfi, Formal Development of Java Based Web Parallel Applications, in: Proc. Hawaii International Conference of System Science (Hawaii, USA, 1998).

[6] G. DiMarzo, N. Guelfi, A. Romanovsky and A.F. Zorzo, Formal Development and Validation of DSGamma System Based on CO-OPN/2 and Coordinated Atomic Actions, Tech. Report, DeVa ESPRIT Project, Newcastle University, 1997.

[7] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques* (Kaufman Publishers, San Mateo, California, USA, 1993).

[8] S.T. Gregory and J.C. Knight, On the provision of Backward Error Recovery in production programming languages, in: Proc. 19th Int. Symp. on Fault-Tolerant Computing (Chicago, Illinois, 1989) 507-511.

[9] C.A.R. Hoare, Monitors - an operating system structuring concept, *Com. of ACM* 17, 10 (1974) 549-557.

[10] C.A.R. Hoare, Parallel Programming: an Axiomatic Approach, in: G. Goos and J. Hartmaur, eds., *Lecture Notes in Computer Science*, LNCS-46 (Springer-Verlag, 1976) 11-39.

[11] P. Jalote and R. Campbell, Atomic actions in concurrent systems, in: Proc. 5th Conference on Distributed Computer Systems (1985) 184-191.

[12] The Java Language Specification. Sun Microsystems, Inc., 1996.

[13] Java Remote Method Invocation Specification. Sun Microsystems, Inc., 1996.

[14] P.A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice* (Springer-Verlag, Wien - New York, 1990).

[15] D.B. Lomet, Process Structuring, Synchronization, and Recovery Using Atomic Actions, *ACM SIGPLAN Notices* 12 (1977).
[16] D.L. Parnas, P.C. Clements and D.M. Weiss, The Modular Structure of Complex Systems, *IEEE TSE-11*, 3 (1985) 259-267.
[17] B. Randell, System structure for software fault tolerance, *IEEE TSE-1*, 2 (1975) 220-232.
[18] B. Randell, A. Romanovsky, R.J. Stroud, J. Xu and A.F. Zorzo, Co-ordinated Atomic Actions: from Concept to Implementation, TR-595, Dept. of Comp. Science, Newcastle University, 1997.
[19] A. Romanovsky, B. Randell, R. Stroud, J. Xu and A. Zorzo, Implementation of blocking Coordinated Atomic Actions based on forward error recovery, *J. of Syst. Arch.* 43, 10 (1997) 687-699.
[20] A. Romanovsky, J. Xu and B. Randell, Exception handling and resolution in distributed object-oriented systems, in: Proc. 16th Int. Conf. on Distributed Computing Systems (Hong Kong, 1996) 545-553.
[21] A. Romanovsky and A.F. Zorzo, On Distribution of Coordinated Atomic Actions, *ACM Oper. Syst. Rev.* 31, 5 (1997) 70-78.
[22] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud and Z. Wu, Fault tolerance in concurrent object-oriented software through coordinated error recovery, in: Proc. 25th Int. Symp. on Fault-Tolerant Computing (USA, 1995) 499-508.
[23] A. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. Stroud and I. Welch, Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: the Production Cell Case Study, Internal Tech. Report, Dept. of Comp. Science, Newcastle University, 1997.